COMP 322: Fundamentals of Parallel Programming

Lecture 38: Course Review

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

HJ isolated statement (Lectures 19, 20)

isolated <body>

- Isolated statement identifies a critical section
- Two tasks executing isolated statements must perform them in mutual exclusion
  - ➔Isolation guarantee applies to (isolated, isolated) pairs of statement instances, not to (isolated, non-isolated) pairs of statement instances
- Isolated statements may be nested
  - An inner isolated statement is redundant
- Parallel constructs should be avoided inside isolated statements
  - Isolated statements must not contain any other parallel statement that performs a blocking operation: finish, future get, next, async await
  - Non-blocking async operations are permitted, but isolation guarantee only applies to creation of async, not to its execution
- Isolated statements can never cause a deadlock
  - Other techniques used to enforce mutual exclusion (e.g., locks) can lead to a deadlock, if used incorrectly

Parallel Spanning Tree Algorithm using isolated statement

```
class V  {
  V [] neighbors; // adjacency list for input graph
  V parent; // output value of parent in spanning tree
  boolean tryLabeling(V n) {
    isolated if (parent == null) parent=n;
    return parent == n; // return true for success
  } // tryLabeling
  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      V child = neighbors[i];
      if (child.tryLabeling(this))
          async child.compute(); //escaping async
     }
  } // compute
} // class V
. . .
root.parent = root; // Use self-cycle to identify root
finish root.compute();
. . .
```

Example graph (root=1, spanning tree edge shown as arrow from child to parent)

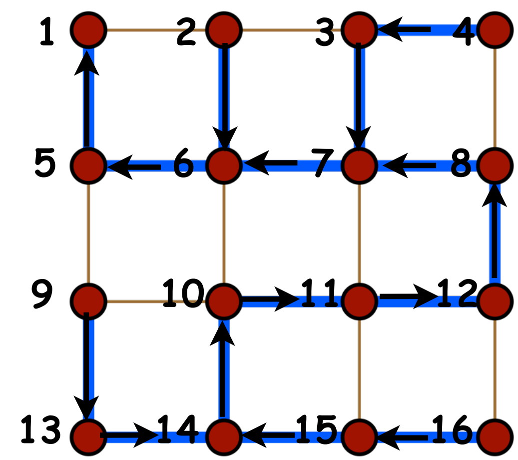
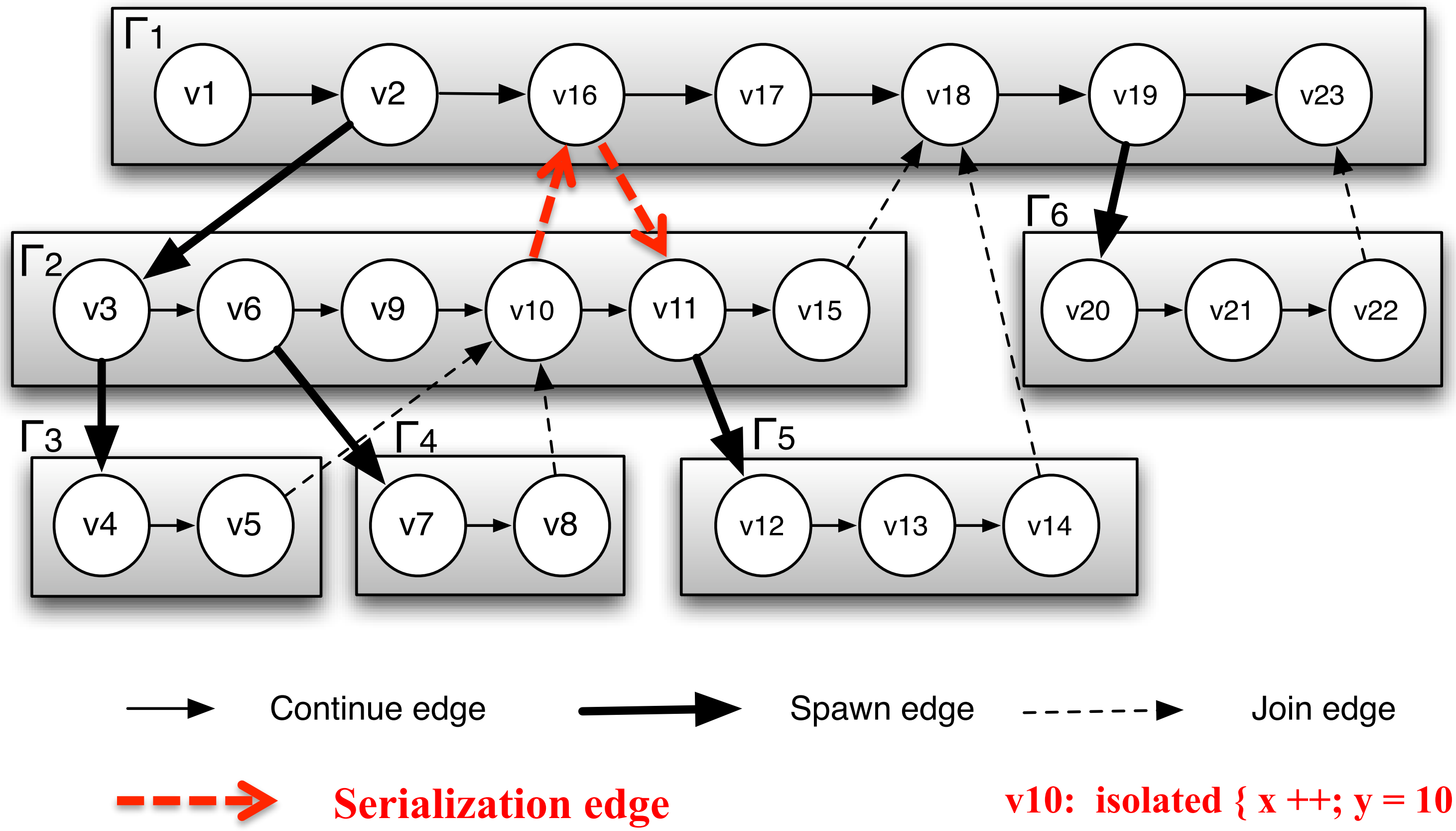


Figure source: http://en.wikipedia.org/wiki/Spanning_tree

Serialized Computation Graph for Isolated Statements

- **Model each instance of an isolated statement as a distinct step (node) in the CG.**
- **Need to reason about the *order* in which interfering isolated statements are executed**
  - **—Complicated because the order of isolated statements may vary from execution to execution**
- **Introduce Serialized Computation Graph (SCG) that includes a specific ordering of all interfering isolated statements.**
  - **—SCG consists of a CG with additional serialization edges.**
  - **—Each time an isolated step, S′, is executed, we add a serialization edge from S to S′ for each prior "interfering" isolated step, S**
    - **Two isolated statements always interfere with each other**
    - **Interference of "object-based isolated" statements depends on intersection of object sets**
    - **Serialization edge is not needed if S and S' are already ordered in CG**
  - **—An SCG represents a set of executions in which all interfering isolated statements execute in the same order.**

Example of Serialized Computation Graph with Serialization Edges for v10-v16-v11 order



v10: isolated { x ++; y = 10; }
v11: isolated { x++; y = 11; }
v16: isolated { x++; y = 16; }

Data race definition can be applied to Serialized Computation Graphs (SCGs) just like regular CGs

— **Need to consider all possible orderings of interfering isolated statements to establish data race freedom**

Object-based isolation in HJ

```
isolated(obj1, obj2, ...) <body>
```

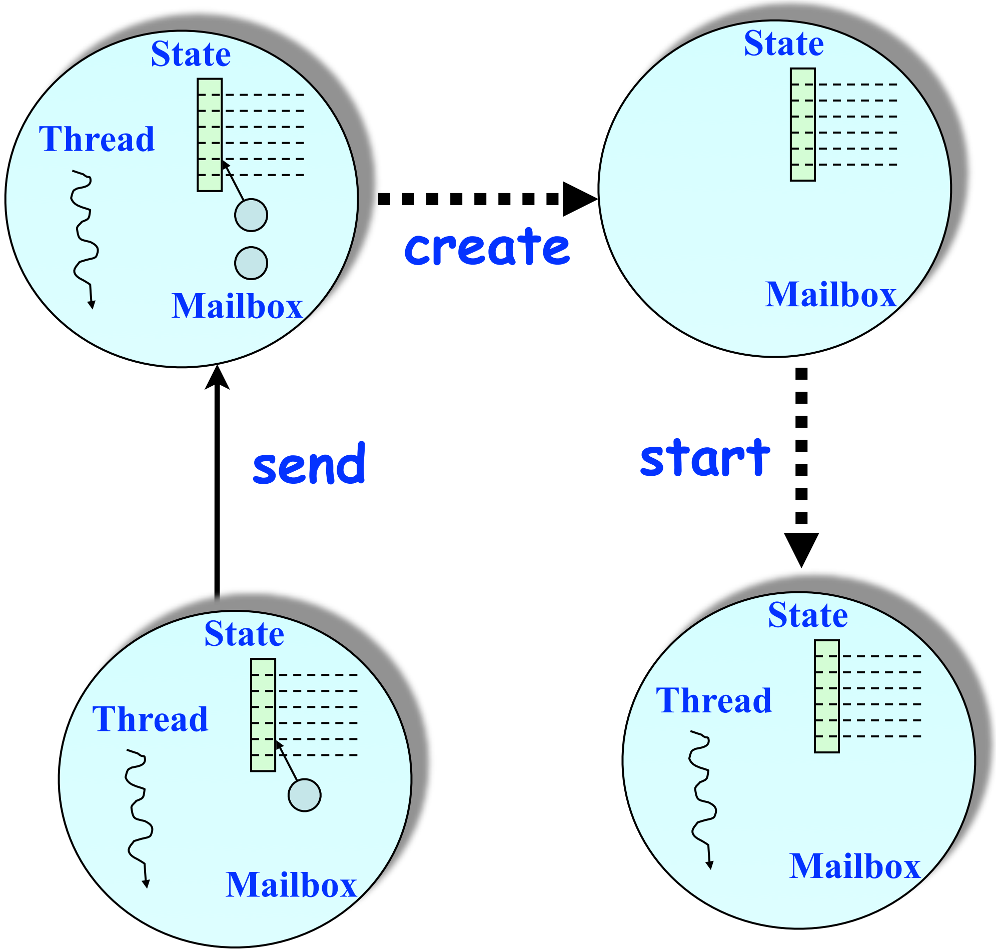
- In this case, programmer specifies list of objects for which isolation is required
- Mutual exclusion is only guaranteed for instances of isolated statements that have a non-empty intersection in their object lists
  - Standard isolated is equivalent to "isolated(*)" by default i.e., isolation across all objects
- Implementation can choose to distinguish between read/write accesses for further parallelism
  - Current HJ implementation supports object-based isolation, but does not exploit read/write distinction

Parallel Spanning Tree Algorithm using Object-based isolation

```
class V  {
  V [] neighbors; // adjacency list for input graph
  V parent; // output value of parent in spanning tree
  boolean tryLabeling(V n) {
    isolated(this) if (parent == null) parent=n;
    return parent == n; // return true for success
  } // tryLabeling
  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      V child = neighbors[i];
      if (child.tryLabeling(this))
          async child.compute(); //escaping async
     }
  } // compute
} // class V
. . .
root.parent = root; // Use self-cycle to identify root
finish root.compute();
. . .
```

The Actor Model (Lectures 21, 22)

- An actor may:
  - —process messages
  - —read/write local state
  - —create a new actor
  - —start a new actor
  - —send messages to other actors
  - —terminate
- An actor processes messages sequentially
  - —guaranteed mutual exclusion on accesses to local state



Actor Life Cycle

Actor states

- New: Actor has been created
  - e.g., email account has been created
- Started: Actor can receive and process messages
  - **e.g., email account has been activated**
- Terminated: Actor will no longer processes messages
  - **e.g., termination of email account after graduation**

Using Actors in HJ

- Create your custom class which extends hj.lang.Actor<Object> ,and implement the void process() method

```
class MyActor extends Actor<Object> {
  protected void process(Object message) {
    System.out.println("Processing " + message);
} }
```

- Instantiate and start your actor

```
Actor<Object> anActor = new MyActor(); anActor.start()
```

- Send messages to the actor

```
anActor.send(aMessage); //aMessage can be any object in general
```

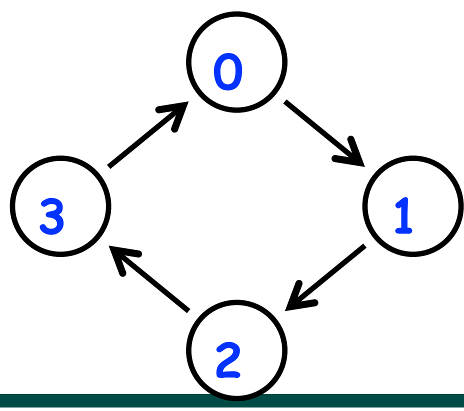
- Use a special message to terminate an actor

```
protected void process(Object message) {
  if (message.someCondition()) exit();
}
```

- **Actor execution implemented as async tasks in HJ**
  - **Can use finish to await completion of an actor!**

ThreadRing (Coordination) Example

```java
finish {
  int numThreads = 4;
  int numberOfHops = 10;
  ThreadRingActor[] ring =
    new ThreadRingActor[numThreads];
  for(int i=numThreads-1;i>=0; i--) {
    ring[i] = new ThreadRingActor(i);
    ring[i].start();
    if (i < numThreads - 1) {
      ring[i].nextActor(ring[i + 1]);
  } }
  ring[numThreads-1].nextActor(ring[0]);
  ring[0].send(numberOfHops);
} // finish
```




```java
class ThreadRingActor
    extends Actor<Object> {
  private Actor<Object> nextActor;
  private final int id;
  ...
  public void nextActor(
    Actor<Object> nextActor) {...}
  void process(Object theMsg) {
    if (theMsg instanceof Integer) {
      Integer n = (Integer) theMsg;
      if (n > 0) {
        println("Thread-" + id +
          " active, remaining = " + n);
        nextActor.send(n - 1);
      } else {
        println("Exiting Thread-"+ id);
        nextActor.send(-1);
        exit();
      }
    } else {
    /* ERROR - handle appropriately */
} } }
```

Summary of Mutual Exclusion approaches in HJ

- Isolated --- analogous to critical sections
- Object-based isolation, isolated(a, b, ...)
  - Single object in list --- like monitor operations on object
  - Multiple objects in list --- deadlock-free mutual exclusion on sets of objects
- Java atomic variables --- optimized implementation of object-based isolation
- Java concurrent collections --- optimized implementation of monitors
- Actors --- different paradigm from task parallelism (mutual exclusion by default)

Linearizability of Concurrent Objects (Lectures 22, 23)

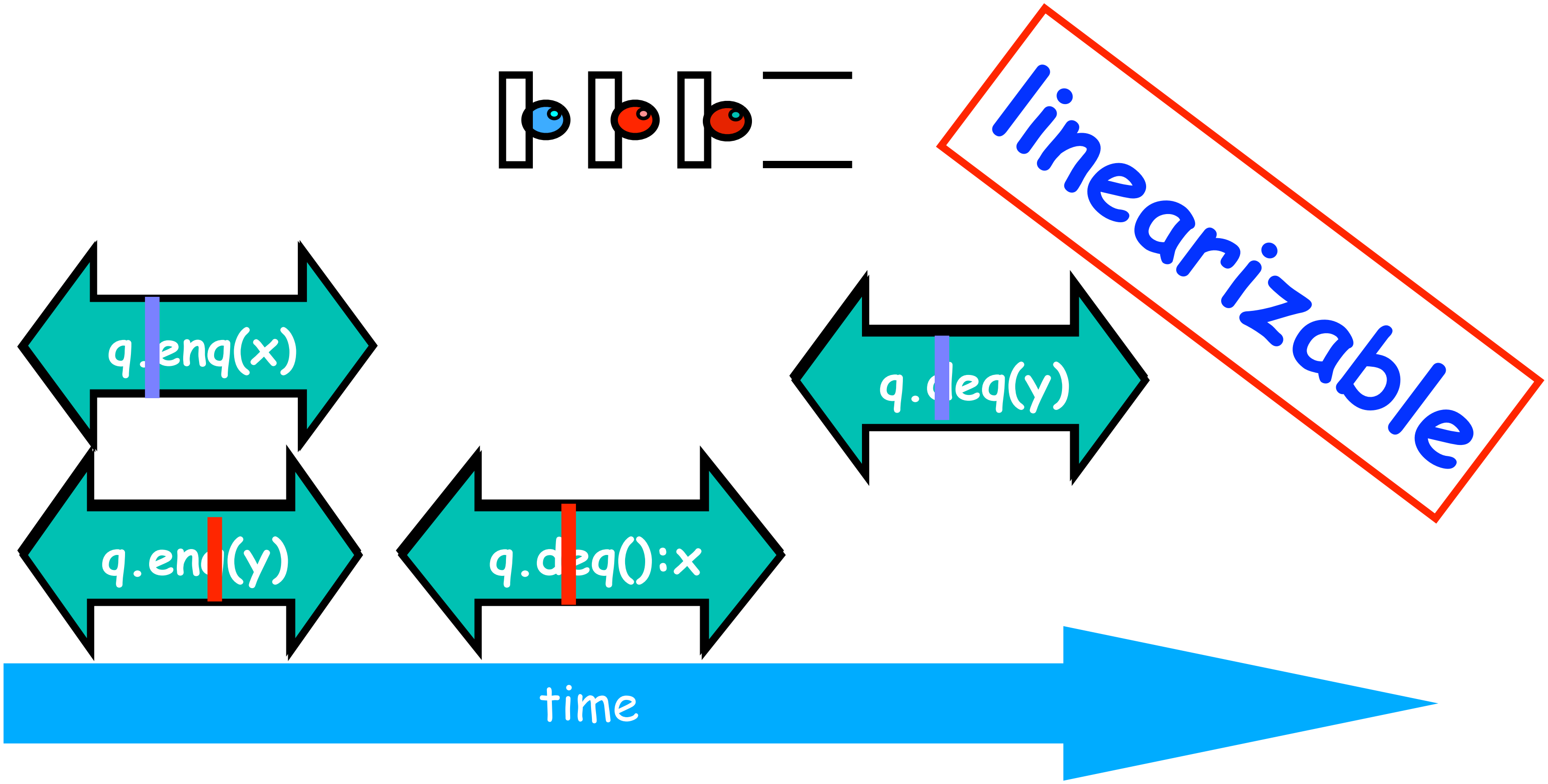
<u>**Concurrent object**</u>

- **A concurrent object is an object that can correctly handle methods invoked in parallel bylin different tasks or threads**
  - **—Examples: concurrent queue, AtomicInteger**

<u>**Linearizability**</u>

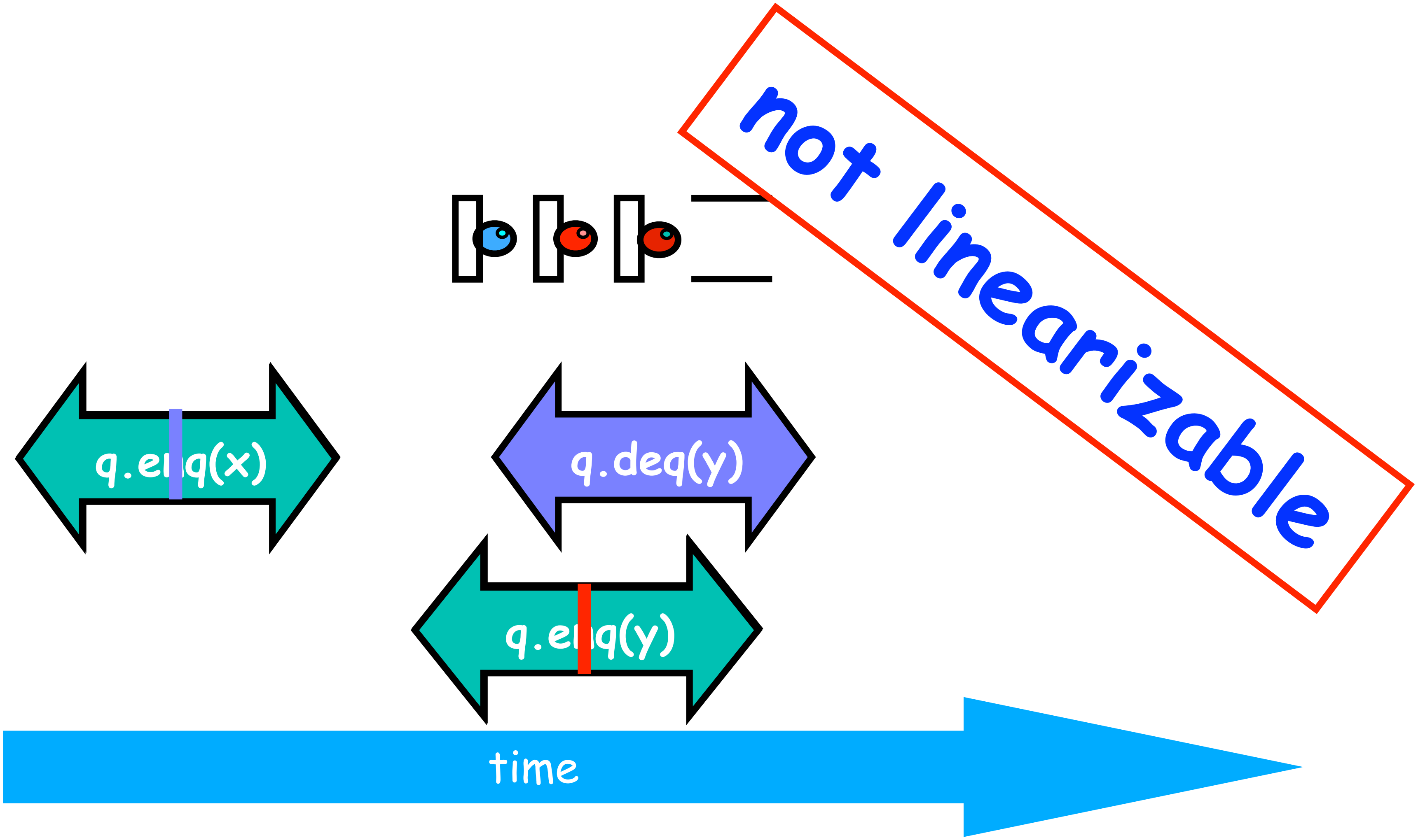
- **Assume that each method call takes effect “instantaneously” at some distinct point in time between its invocation and return.**
- **An <u>execution</u> is linearizable if we can choose instantaneous points that are consistent with a sequential execution in which methods are executed at those points**
- **An <u>object</u> is linearizable if all its possible executions are linearizable**

Example 1



Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Example 2



Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Safety vs. Liveness (Lecture 24)

- In a concurrent setting, we need to specify both the safety and the liveness properties of an object
- Need a way to define
  - —Safety: when an implementation is correct
  - —Liveness: the conditions under which it guarantees progress

- Data race freedom is a desirable safety property for most parallel programs
- Linearizability is a desirable safety property for most concurrent objects

Liveness Guarantees

- Liveness = a program's ability to make progress in a timely manner
- Different levels of liveness guarantees (from weaker to stronger)
  - Deadlock freedom
  - Livelock freedom
  - Starvation freedom
  - Bounded wait

Two-way Parallel ArraySum using Java threads (Lecture 24)

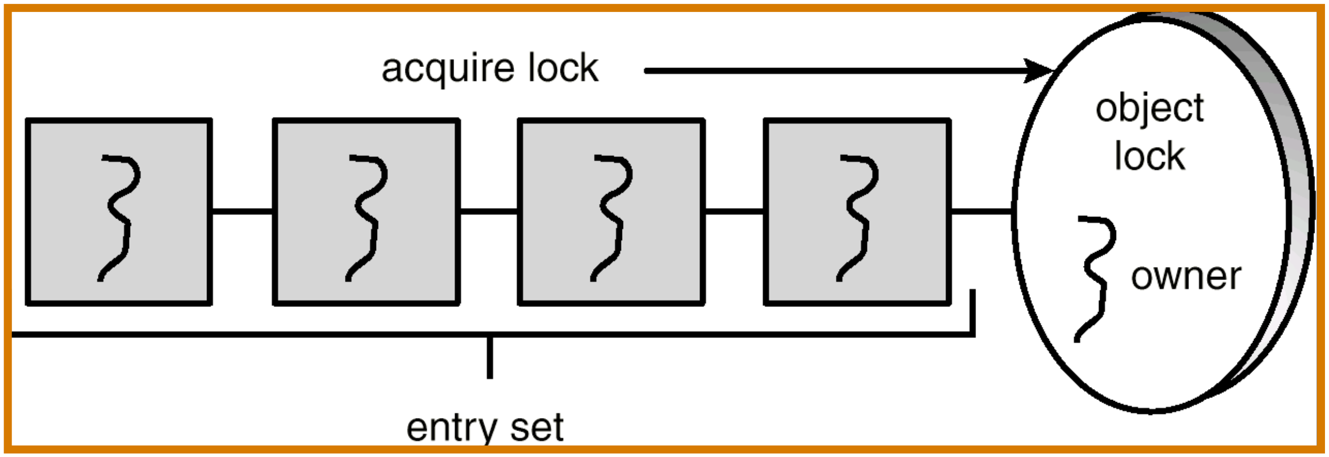
```
// Start of Task T1 (main program)
sum1 = 0; sum2 = 0; // Assume that sum1 & sum2 are fields (not local vars)
// Compute sum1 (lower half) and sum2 (upper half) in parallel
final int len = X.length;
Runnable r1 = new Runnable() {
  public void run(){ for(int i=0 ; i < len/2 ; i++) sum1 += X[i];}
};
Thread t1 = new Thread(r1);
t1.start();
Runnable r2 = new Runnable() {
  public void run(){ for(int i=len/2 ; i < len ; i++) sum2 += X[i];}
};
Thread t2 = new Thread(r2);
t2.start();
// Wait for threads t1 and t2 to complete
t1.join(); t2.join();
int sum = sum1 + sum2;
```

Objects and Locks in Java --- synchronized statements and methods (Lecture 25)

- Every Java object has an associated *lock* acquired via:
  - `synchronized` statements
    - synchronized(foo) { // acquire foo's lock
      // execute code while holding foo's lock
      } // release foo's lock
  - `synchronized` methods
    - public synchronized void op1() { // acquire 'this' lock
      // execute method while holding 'this' lock
      } // release 'this' lock
- Java language does not enforce any relationship between object used for locking and objects accessed in isolated code
  - If same object is used for locking and data access, then the object behaves like a monitor
- Locking and unlocking are automatic
  - Locks are released when a synchronized block exits
    - By normal means: end of block reached, return, break
    - When an exception is thrown and not caught

Implementation of Java synchronized statements/methods

- Every object has an associated lock
- "synchronized" is translated to matching monitorenter and monitorexit bytecode instructions for the Java virtual machine
  - —monitorenter requests "ownership" of the object's lock
  - —monitorexit releases "ownership" of the object's lock
- If a thread performing monitorenter does not own the lock (because another thread already owns it), it is placed in an unordered "entry set" for the object's lock



java.util.concurrent.locks.Lock interface (Lecture 26)

```
interface Lock {
   void lock();
   void lockInterruptibly() throws InterruptedException;
   boolean tryLock(); // return false if lock is not obtained
   boolean tryLock(long timeout, TimeUnit unit)
                                throws InterruptedException;
   void unlock();
   Condition newCondition();
   // can associate multiple condition vars with lock
 }
```

- **java.util.concurrent.locks.Lock interface is implemented by java.util.concurrent.locks.ReentrantLock class**

java.util.concurrent.locks.ReadWriteLock interface

```
interface ReadWriteLock {
  Lock readLock();
  Lock writeLock();
}
```

- Even though the interface appears to just define a pair of locks, the semantics of the pair of locks is coupled as follows
  - Case 1: a thread has successfully acquired writeLock().lock()
    - No other thread can acquire readLock() or writeLock()
  - Case 2: no thread has acquired writeLock().lock()
    - Multiple threads can acquire readLock()
    - No other thread can acquire writeLock()
- java.util.concurrent.locks.ReadWriteLock interface is implemented by java.util.concurrent.locks.ReadWriteReentrantLock class

Java Executors and Synchronizers (Lecture 28)

- **Atomic variables**
  - —The key to writing lock-free algorithms
- **Concurrent Collections:**
  - —Queues, blocking queues, concurrent hash map, …
  - —Data structures designed for concurrent environments
- **Locks and Conditions**
  - —More flexible synchronization control
  - —Read/write locks
- **Executors, Thread pools and Futures**
  - —Execution frameworks for asynchronous tasking
- **Synchronizers: Semaphore, Latch, Barrier, Exchanger**
  - —Ready made tools for thread coordination

Summary: Relating j.u.c. libraries to HJ constructs

- Atomics: java.util.concurrent.atomic

  Can be used as is in HJ programs

- Concurrent Collections

  Can be used as is in HJ programs

- Locks: java.util.concurrent.locks

  Many uses of j.u.c.locks & synchronized can be replaced by HJ isolated

- Synchronizers

  Many uses can be replaced by finish, phasers, and data-driven futures

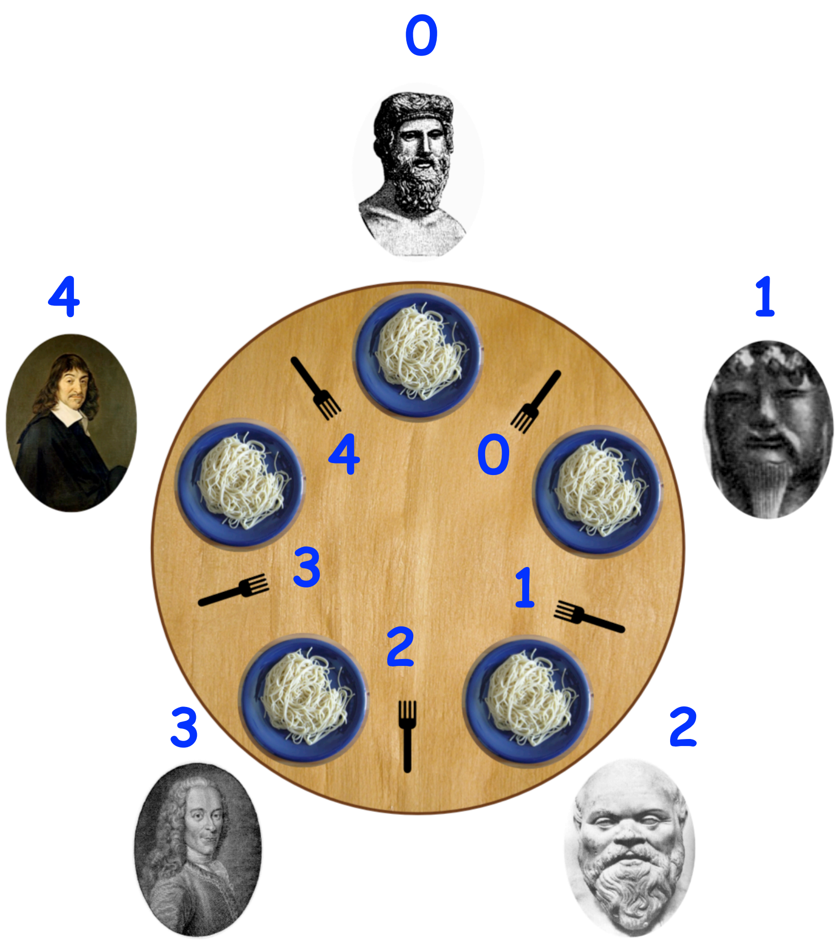
- Executors

  Many uses can be replaced by async, finish, futures, forall

- Queues

  Do not use BlockingQueue in HJ programs, and take care to avoid infinite loops on retrieval operations on non-blocking queues

The Dining Philosophers Problem (Lecture 29)



Constraints

- Five philosophers either eat or think
- They must have two forks to eat (don't ask why)
- Can only use forks on either side of their plate
- No talking permitted

Goals

- Progress guarantees
  - Deadlock freedom
  - Livelock freedom
  - Starvation freedom
  - Bounded wait
- Maximize concurrency when eating

	Deadlock	Livelock	Starvation	Non-concurrency
Solution 1: synchronized	Yes	No	Yes	Yes
Solution 2: tryLock/ unLock	No	Yes	Yes	Yes
Solution 3: isolated	No	No	Yes	Yes
Solution 4: object-based isolation	No	No	Yes	No
Solution 5: semaphores	No	No	No	No
Solution 6: actors	No	Yes	Yes	No

Places in HJ (Lecture 30)

here = place at which current task is executing

place.MAX_PLACES = total number of places (runtime constant)

Specified by value of **p** in runtime option, `-places p:w`

place.factory.place(i) = place corresponding to index i

<place-expr>.toString() returns a string of the form “place(id=0)”

<place-expr>.id returns the id of the place as an int

async at(P) S

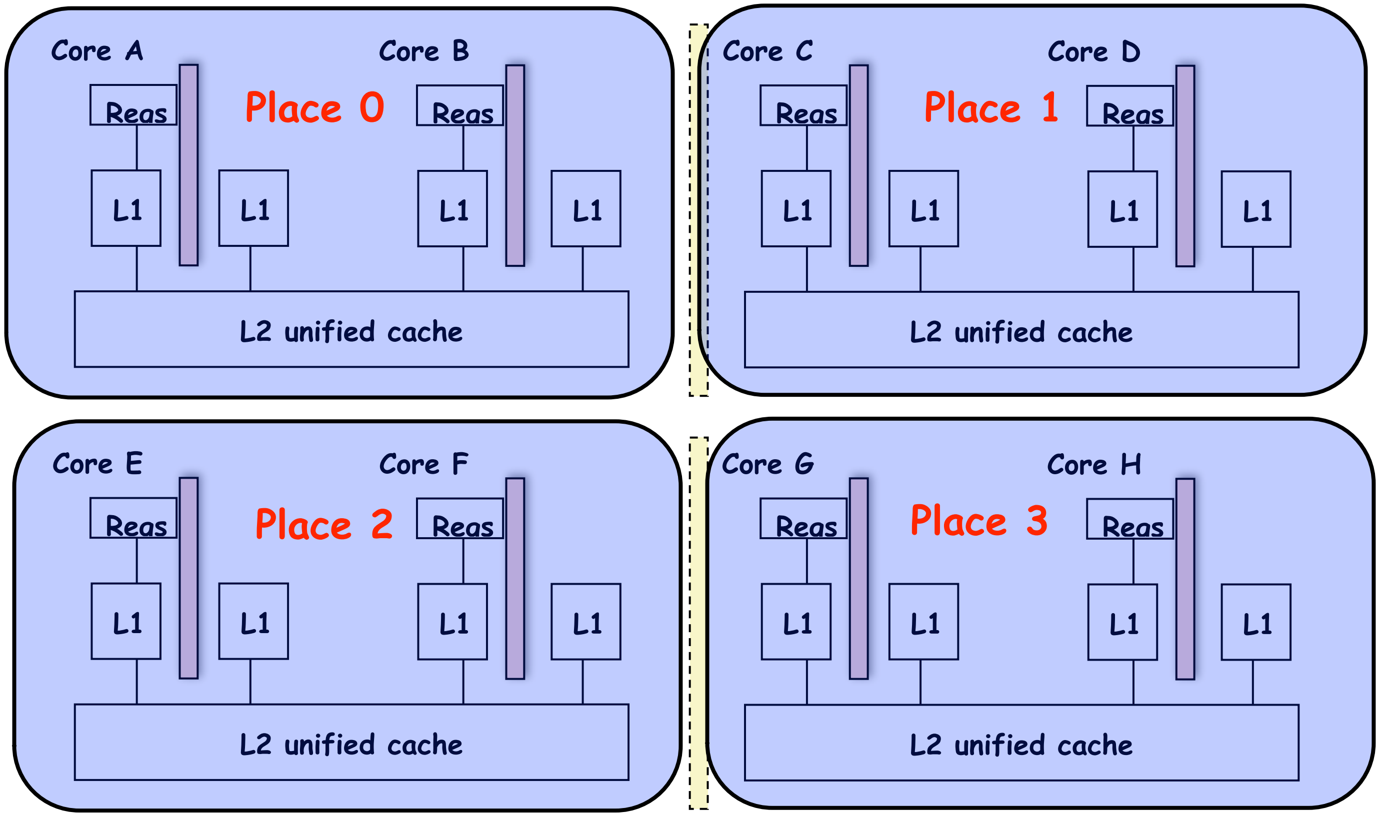
- Creates new task to execute statement S at place P
- **async S** is equivalent to **async at(here) S**
- Main program task starts at **place.factory.place(0)**

Note that **here** in a child task refers to the place P at which the child task is executing, not the place where the parent task is executing

Example of –places 4:2 option on an 8-core node (4 places w/ 2 workers per place)

```
// Main program starts at place 0
async at(place.factory.place(0)) S1;
async at(place.factory.place(0)) S2;
```

```
async at(place.factory.place(1)) S3;
async at(place.factory.place(1)) S4;
async at(place.factory.place(1)) S5;
```




```
async at(place.factory.place(2)) S6;
async at(place.factory.place(2)) S7;
async at(place.factory.place(2)) S8;
```

```
async at(place.factory.place(3)) S9;
async at(place.factory.place(3)) S10;
```

Example HJ program with places

```
class T1 {
  final place affinity;
  . . .
  // T1's constructor sets affinity to place where instance was created
  T1() { affinity = here; ... }
  . . .
}
. . .
finish { // Inter-place parallelism
  System.out.println("Parent place = ", here); // Parent task s place
  for (T1 a = . . .) {
    async at (a.affinity) { // Execute async at place with affinity to a
      a.foo();
      System.out.println("Child place = ", here); // Child task's place
    } // async
  } // for
} // finish
. . .
```

Adding support for places in HJ actors (Lecture 31)

- **Basic approach: include an optional place parameter in the start() method**

  Actor<Object> anActor = new MyActor();

  anActor.start(**p**); // Start actor at place p

- **Example:**

  SievePlaceActor nextActor = new SievePlaceActor(...);

  // Start actor at next place, relative to current place

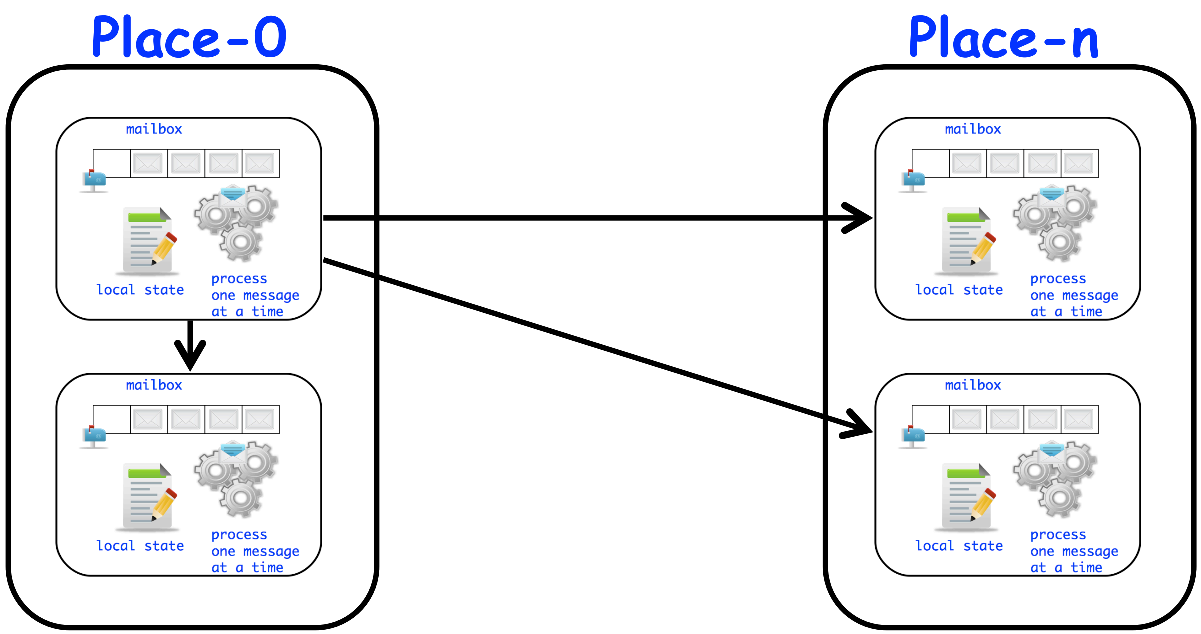
  nextActor.start(**here.next()**);

  // This ensures locality with respect to local primes stored

COMP 322, Spring 2013 (V. Sarkar)

Actor and Places

- Places act as containers for Actors
- Actors from different places can send each other messages
- Actor always processes the message in a specified place
  - Easier to achieve data locality via local state



COMP 322, Spring 2013 (V. Sarkar)

Introduction to MPI (Lectures 32, 33, 34)

main() is enclosed in an implicit "forall" --- each process runs a separate instance of main() with "index variable" = myrank

```
import mpi.*;
class Hello {
    static public void main(String[] args) {
        // Init() be called before other MPI calls
        MPI.Init(args); /
        int npes = MPI.COMM_WORLD.Size()
        int myrank = MPI.COMM_WORLD.Rank() ;
        System.out.println("My process number is " + myrank);
        MPI.Finalize(); // Shutdown and clean-up
    }
}
```

Example with Send and Recv

```
import mpi.*;

class myProg {
  public static void main( String[] args ) {
    int tag0 = 0;
    MPI.Init( args );                   // Start MPI computation
    if ( MPI.COMM_WORLD.rank() == 0 ) { // rank 0 = sender
      int loop[] = new int[1]; loop[0] = 3;
      MPI.COMM_WORLD.Send( "Hello World!", 0, 12, MPI.CHAR, 1, tag0 );
      MPI.COMM_WORLD.Send( loop, 0, 1, MPI.INT, 1, tag0 );
    } else {                            // rank 1 = receiver
      int loop[] = new int[1]; char msg[] = new char[12];
      MPI.COMM_WORLD.Recv( msg, 0, 12, MPI.CHAR, 0, tag0 );
      MPI.COMM_WORLD.Recv( loop, 0, 1, MPI.INT, 0, tag0 );
      for ( int i = 0; i < loop[0]; i++ ) System.out.println( msg );
    }
    MPI.Finalize( );                    // Finish MPI computation
  }
}
```

Send() and Recv() calls are blocking operations by default

Approach #1 to Deadlock Avoidance --- Reorder Send and Recv calls

We can break the circular wait to avoid deadlocks as follows:

```
int a[], b[];
...
if (MPI.COMM_WORLD.rank() == 0) {
    MPI.COMM_WORLD.Send(a, 0, 10, MPI.INT, 1, 1);
    MPI.COMM_WORLD.Send(b, 0, 10, MPI.INT, 1, 2);
}
else {
    Status s1 = MPI.COMM_WORLD.Recv(a, 0, 10, MPI_INT, 0,
1);
    Status s2 = MPI.COMM_WORLD.Recv(b, 0, 10, MPI.INT, 0,
2);
}
...
```

Using Sendrecv for Deadlock Avoidance in Scenario #2

Consider the following piece of code, in which process i sends a message to process i + 1 (modulo the number of processes) and receives a message from process i - 1 (modulo the number of processes)

```
int a[], b[];
. . .
int npes = MPI.COMM_WORLD.size();
int myrank = MPI.COMM_WORLD.rank()
MPI.COMM_WORLD.Sendrecv(a, 0, 10, MPI.INT, (myrank+1)%npes, 1,
                        b, 0, 10, MPI.INT, (myrank-1+npes)%npes,
1);

...
```

A combined Sendrecv() call avoids deadlock in this case

Simple Irecv() example

- **The simplest way of waiting for completion of a single non-blocking operation is to use the instance method Wait() in the Request class, e.g:**

```
// Post a receive operation
Request request = Irecv(intBuf, 0, n, MPI.INT,
                        MPI.ANY_SOURCE,  0) ;
// Do some work while the receive is in progress
…
// Finished that work, now make sure the message has
  arrived
Status status = request.Wait() ;
// Do something with data received in intBuf
…
```

- **The Wait() operation is declared to return a Status object. In the case of a non-blocking receive operation, this object has the same interpretation as the Status object returned by a blocking Recv() operation.**

Collective Communications

- **Each collective operation is defined over a communicator (most often, MPI.COMM_WORLD)**
  - **Each collective operation contains an *implicit barrier*. The operation completes and execution continues when all processes in the communicator perform the *same* collective operation.**
  - **A mismatch in operations results in *deadlock* e.g.,**

    Process 0: .... MPI.Bcast(...) ....

    Process 1: .... MPI.Bcast(...) ....

    Process 2: .... MPI.Gather(...) ....
- **We can model the synchronization performed by MPI operations as phasers to understand their semantics**
  - **Assume that all processes are registered on multiple phasers, one for each kind of collective operation e.g., ph1 for Bcast, ph2 for Gather**
  - **The above example can be rewritten as follows, where doNext() performs a "next" operation on one phaser only**

    Process 0: .... ph1.doNext(); ....

    Process 1: .... ph1.doNext(); ....

    Process 2: .... ph2.doNext(); ....

Examples of Collective Operations

```
void Barrier()
```

– **Blocks the caller until all processes in the group have called it.**

```
void Gather(Object sendbuf, int sendoffset, int sendcount,
  Datatype sendtype, Object recvbuf, int recvoffset,
  int recvcount, Datatype recvtype, int root)
```

– **Each process sends the contents of its send buffer to the root process.**

```
void Scatter(Object sendbuf, int sendoffset, int sendcount,
  Datatype sendtype, Object recvbuf, int recvoffset,
  int recvcount, Datatype recvtype, int root)
```

– **Inverse of the operation Gather.**

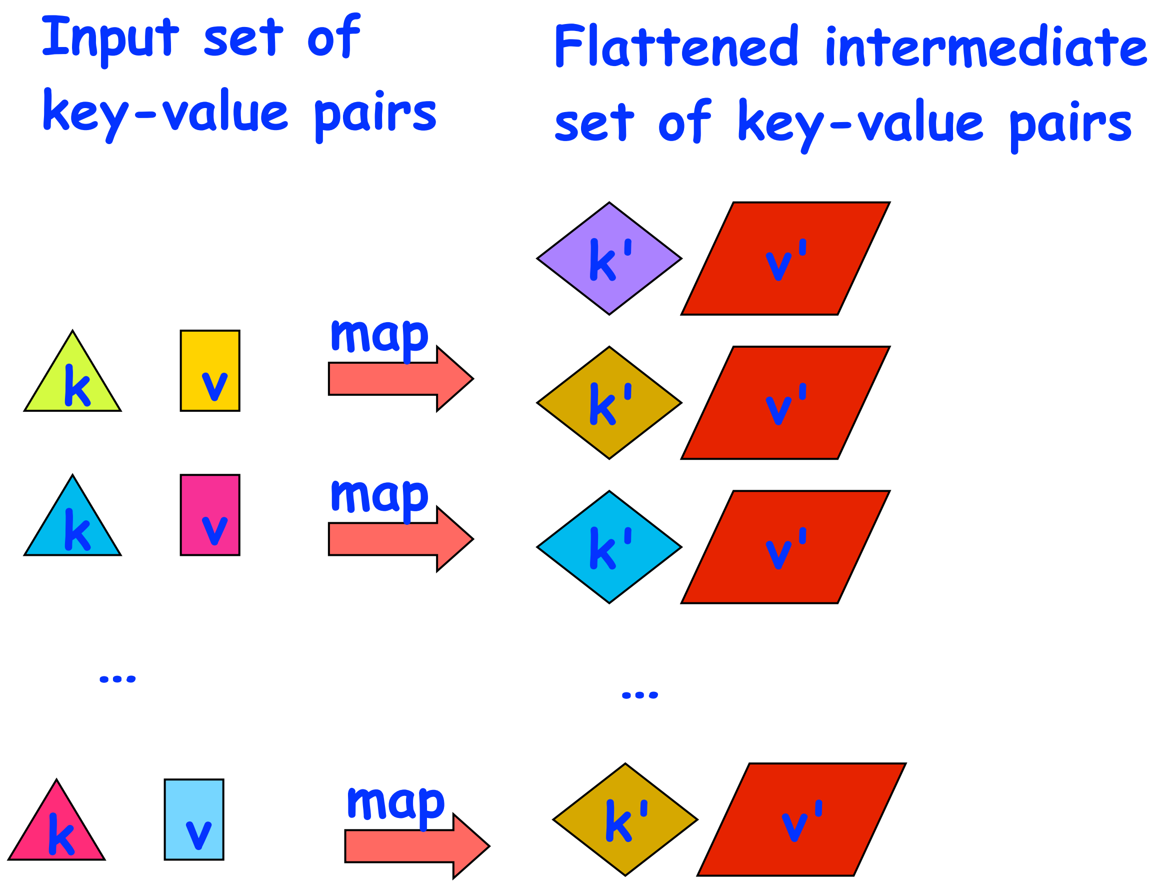
```
void Reduce(Object sendbuf, int sendoffset, Object recvbuf,
  int recvoffset, int count, Datatype datatype, Op op,
  int root)
```

– **Combine elements in send buffer of each process using the reduce operation, and return the combined value in the receive buffer of the root process.**

Operations on Sets of Key-Value Pairs (Lecture 35)

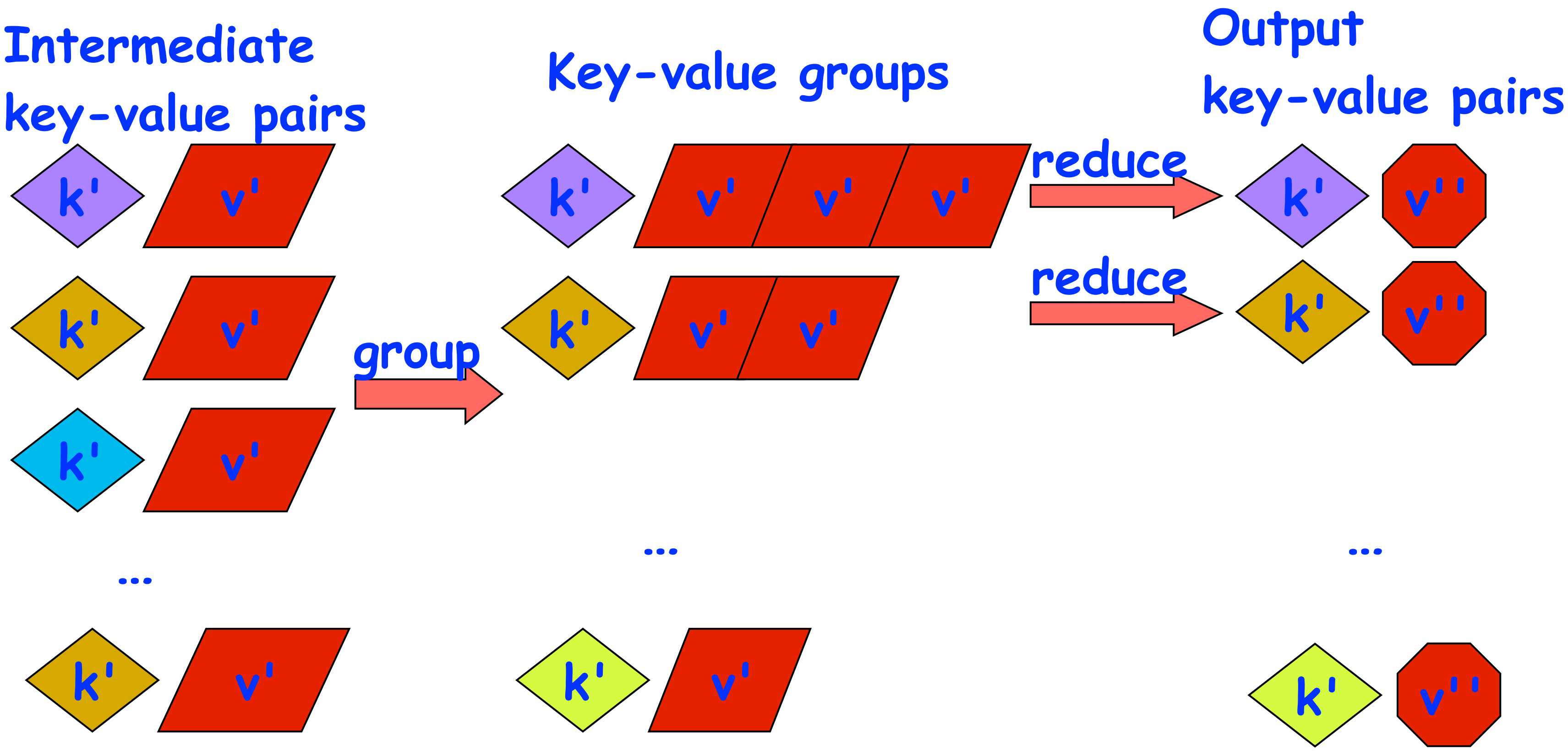
- Input set is of the form {(k1, v1), . . . (kn, vn)}, where (ki, vi) consists of a key, ki, and a value, vi.
  - Assume that the key and value objects are immutable, and that equality comparison is well defined on all key objects.
- Map function f generates sets of intermediate key-value pairs, f(ki,vi) = {(k1′ ,v1′),...(km′,vm′)}. The kj′ keys can be different from ki key in the input of the map function.
  - Assume that a flatten operation is performed as a post-pass after the map operations, so as to avoid dealing with a set of sets.
- Reduce operation groups together intermediate key-value pairs, {(k′, vj′)} with the same k', and generates a reduced key-value pair, (k′,v″), for each such k', using reduce function g

MapReduce: The Map Step



Source: http://infolab.stanford.edu/~ullman/mining/2009/mapreduce.ppt

MapReduce: The Reduce Step

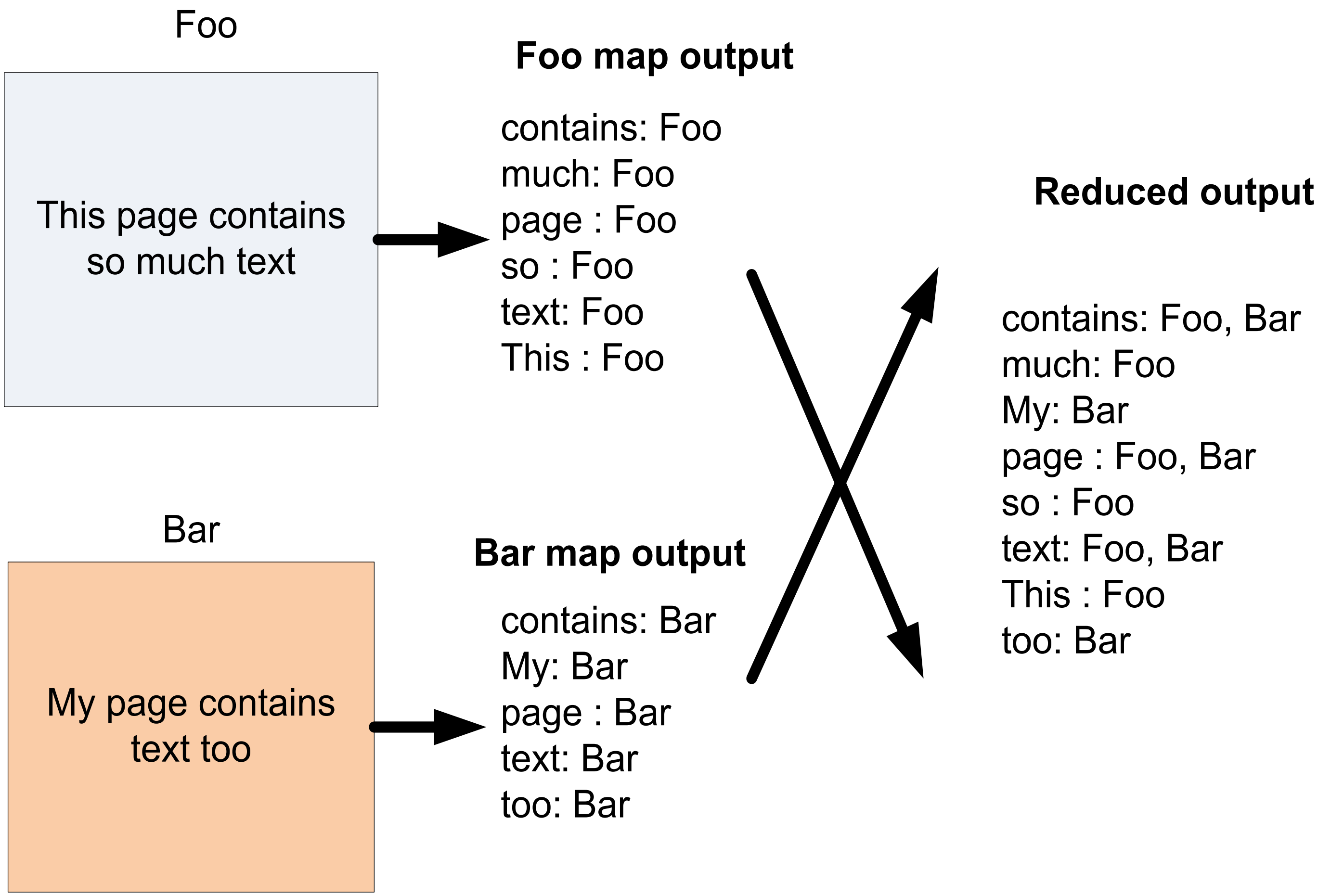


Source: http://infolab.stanford.edu/~ullman/mining/2009/mapreduce.ppt

Algorithms for MapReduce

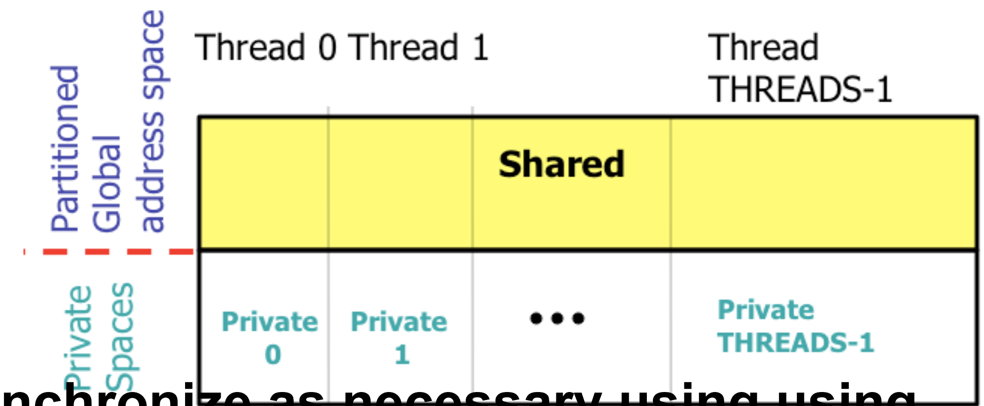
- <u>Sorting</u>
- Searching
- <u>Indexing</u>
- Classification
- <u>TF-IDF</u>
- <u>Breadth-First Search / SSSP</u>
- PageRank
- Clustering

Inverted Index: Data flow



UPC Execution Model (Lecture 36)

- Multiple threads working independently in a SPMD fashion
  - —MYTHREAD specifies thread index (0..THREADS-1)
    - Like MPI processes and ranks
  - —# threads specified at compile-time or program launch
- Partitioned Global Address Space (different from MPI)



- Threads synchronize as necessary using using
  - —synchronization primitives
  - —shared variables

Worksheet #36: UPC data distributions

In the following example from slide 23, assume that each UPC array is distributed by default across threads with a cyclic distribution. In the space below, identify an iteration of the upc_forall construct for which all array accesses are local, and an iteration for which all array accesses are non-local (remote). Explain your answer in each case.

```
shared int a[100],b[100], c[100];
int i;
upc_forall (i=0; i<100; i++; (i*THREADS)/100)
   a[i] = b[i] * c[i];
```

Solution:

- Iteration 0 has affinity with thread 0, and accesses a[0], b[0], c[0], all of which are located locally at thread 0
- Iteration 1 has affinity with thread 0, and accesses a[1], b[1], c[1], all of which are located remotely at thread 1

Comparison of Multicore Programming Models along Selected Dimensions

	Dynamic Parallelism	Locality Control	Mutual Exclusion	Collective & Point-to-point Synchronization	Data Parallelism
Cilk	Spawn, sync	None	Locks	None	None
Java Concurrency	Executors, Task Queues	None	Locks, monitors, atomic classes	Synchronizers	Concurrent collections
Intel C++ Threading Building Blocks	Generic algorithms, tasks	None	Locks, atomic classes	None	Concurrent containers
.Net Parallel Extensions	Generic algorithms, tasks	None	Locks, monitors	Futures	PLINQ
OpenMP	SPMD (v2.5), Tasks (v3.0)	None	Locks, critical, atomic	Barriers	None
CUDA	None until recently (v5)	Device, grid, block, threads	None	Barriers	SPMD
Habanero-Java (builds on Java Concurrency)	Async, finish	Places	Isolated blocks, Java atomic classes	Phasers, futures, data-driven tasks	Parallel array operations, Java concurrent collections

Announcements (Recap)

- Graded midterm exams can be picked up from Sherry Nassar in Duncan Hall 3139
- Homework 6 is officially due today, but everyone can get an automatic penalty-free extension till April 26th
  - —No need to send a request for this extension
- Final exam will be given today to be taken in any two-hour duration returned to Sherry Nassar by April 26th (as was done with midterm exams)
  - — Final exam will cover material from Lectures 19 - 36
- Today is the last lecture!

Acknowledgments

- **Graduate TAs**
  - Kumud Bhandari
  - Deepak Majeti
  - Sriraj Paul
  - Rishi Surendran
- **Undergraduate TAs**
  - Annirudh Prasad
  - Yunming Zhang
- **HJ consultants**
  - Vincent Cave
  - Max Grossman
  - Shams Imam
- **Administrative assistant**
  - Sherry Nassar

Have a great summer!!